CS2030S Final

AY24/25 sem 2

github.com/mendax1234

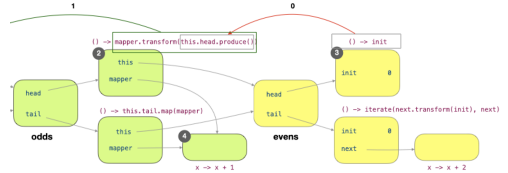
First Half

1. **Information Hiding**:
   - **fields** should be declared as private
   - **methods** should be declared as public
2. **final**: final in a **method declaration** prevents **overriding**.
3. **Heap and Stack**
   - **Stack**: Stack contains stack frames, which are created whenever a **method** is called. It contains (**From bottom to up**)
     - the this reference (If **instance method** is called)
     - the method arguments
     - local variables within the method.
   - **Heap**: Contains the following (**From up to bottom**):
     - **Class name**. (If a **lambda expression**, then write the expression)
     - Instance fields and respective fields.
     - Captured values. (Separated by **dashed lines**)
4. **Override vs. Overload**
   - **Override**: must have **same method descriptor (method signature + method return type)**, e.g.A C::foo(B1,B2), (B1, B2 are the type of the method parameters, same for as follows)
   - **Overload**: must have same **method name**, in the same class and **different method signature (method name, number of parameters, type of each parameter, order of the parameters)**. e.g. C::foo(B1,B2). **The return type of the method doesn't matter.**
5. **Tell, Don't Ask**: We never **ask** an object to spit out its own **raw data**. Instead, we **let the object know** what we want so that it can give us a piece of **processed data** (via an instance method).
   - **Sample reasoning**: The subclass should ask the super class to do the thing (to be changed).
6. **Liskov Substitution Principle**: A *subclass* **should not** break the expectations / **specifications** set by the **superclass**. **Tips**:
   - Always write down what the specifications are set by the superclass.
   - Construct a method and test whether the subclass can be substituted without breaking the specifications. (If class B **extends** A, and **overrides** the method in A, then **successful substitution** means **when substitute A with B**, we should call the **overridden function in B**!)
7. **Method Invocation**: Pay attention to the **CTT, RTT** of the **target** and the **CTT** of the **parameter**. The **RTT** of the **parameter** doesn't matter!
   - During the compile time, find the **most specific** method descriptor starting from the CTT of the target. (Method M is **more specific than** method N means that the **type of the parameter of M** is the **subtype** of the **type of the parameter in N**).
   - During the run time, use the method descriptor we got from above to find **the first** method from the RTT to Object and execute it.
   - **Type casting** happens during the **compile tile**! e.g. (Circle) o2;, the CTT of the method parameter is **Circle** even if o2 might be an Object.
8. **Variance Relationship**: Let S denote the type of element in the "array". Then the **complex type** have three possible variance relationship:
   - **Covariant**: if S <: T, then C(S)<:C(T).
   - **Contravariant**: if S <: T, then C(T) <: C(S)
   - **Invariant**: it is neither **covariant** nor **contravariant**. e.g. **Java Generics**, if S $<:$ T, A<S> $\not<:$ A<T>.
9. **Wrapper Class**: Integer $\not<:$ Double. Note that int[] $\not<:$ double[].
10. **Type erasure**
    - Replace **generic type** with its **raw type**.
    - Replace type parameters.
      - **Non-bounded type parameters** are replaced with Object
      - **Bounded type parameters** are replaced with **the first bound** and **explicitly cast to the second bound**.
    - Insert necessary cast (Usually narrowing conversion) to make sure casting to the expected type.
    - **Example**: this code <U extends Container> void check(U con) {} will become void check(Container con) {} **after type erasure**.
11. **Type inference**
    - **Rule to find constraints**
      - **Target**: "the **return type** of the method" $<:$ "the type of the variable you are assigning to"
      - **Argument**: "the type of the **argument**" $<:$ "the type of the **parameter**"
      - **Bound**: we need to consider "the **bound of the generic type parameters**"
    - **Rules to solve constraints**
      - Type1<:T<:Type2, then T is inferred as Type1
      - Type1<:T, then T is inferred as Type1
      - T<:Type2, then T is inferred as Type2
    - **Type inference involves wildcard**
      - If parameter type is Seq<? super T>, argument type is Seq<G>, then T<:G
      - If parameter type is Seq<? extends T>, argument type is Seq<G>, then G<:T
    - If class A implements Comparable<A>, and class B extends A, then B actually implements Comparable<A> **not** Comparable<B>!
12. **PECS**: This rule is regard to **method parameter**, not the **method**.
13. **Tips**
    - If you pass an integer 3 to a parameter of type **Double**, the code **won't compile**! No auto-boxing is done here!
    - **Subset thinking for wildcard**: ? extends T is a **set of** type X where $\{X : X <: T\}$. Similar for the other.
    - **EAT thinking in PECS**: Use X, Y to represent the **range** for the parameters, T, U for the original type of parameter. Then draw a set notation to decide **when** X, Y is most flexible!
    - **Unbounded wildcard**: List<?> l; Object o = l.get(0);, can **only assign** to Object!
    - **Subtype reasoning**: Think about whether the **subtype** can substitute the **supertype** successfully.
    - **Generics**: Generics can enforce **compile-time type safety**.

Nested Class

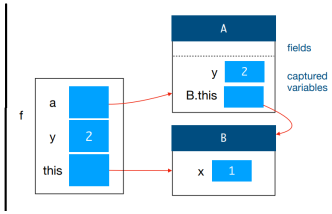
1. **Inner Class**: Suppose A is the **containing class**, B is the **inner class**
   - **Instantiate the inner class**: A.B b = a.new B();
   - **Access instance field x in A from B**: A.this.x=1 this is called **qualified this**.
   - If inner class B is **private**, A.B is **not allowed**! Calling the method inside B is also **not allowed**! (**Apply to static nested class** also!)
2. **Static nested Class**: Same suppose as inner class
   - **Instantiate the inner class**: A.B b = new A.B(); or B b = new B();
   - **Access instance field x in A from B**: Need an instance of A, like A a = new A(); first, then inside the static nested class, can use this.y = a.x + 1; (y is an instance field in B)
   - **this keyword**: this is allowed in the **static nested class** as long as it is used in a **non-static** method.
3. **Anonymous class**:
   - An anonymous class can only extend **one class** or implement **one interface**.
4. **Variable Capture**
   - The **local variables (arguments and normal local variables)** of the **method** where the local class comes from (including all the arguments/variables **that the lambda** uses). The **member/field** of the **local class** (ownself's) is **not captured**!
   - The **instance** that invokes the method where the local class comes from. No members of that instance. No **effective final rule** on the instance. **Update of the member is synced**!
   - Think of Lambda expression as an **anonymous class or local class**, but it has some restrictions on the variables that they can use
     - **Instance or Static Variables (from enclosing class)**: Freely use, no restriction, but actually capture the instance.
     - **Local Variables (the enclosing method)**: Must be effectively final
     - **Parameters of Lambda**: Freely used
     - **Shadowing**: use its own **lambda parameter** instead of the captured variable from enclosing method.
5. **Effectively Final**: An implicitly final variable **cannot be re-assigned** after they are captured, but can be read.
6. **Aliasing**: Key is to judge is **whether two references share the same address**.
7. **Factory method**: Must be static. Otherwise, no way to instantiate an instance.
8. **Varargs ...**: Used for passing in an **array of items (or same type)** to a method.
9. **Stack and Heap**

```
InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2);
InfiniteList<Integer> odds = evens.map(x -> x + 1);
InfiniteList<Integer> altEvens = odds.map(x -> x * 2);
altEvens.head();
```



The **nested class** convention is shown as follows:

```
class B {
  int x = 1;
  void f() {
    int y = 2;
    class A {
      void g() {
        B.this.x = y;
      }
    }
    A a = new A();
    a.g();
  }
}
// main
B b = new B();
b.f();
```



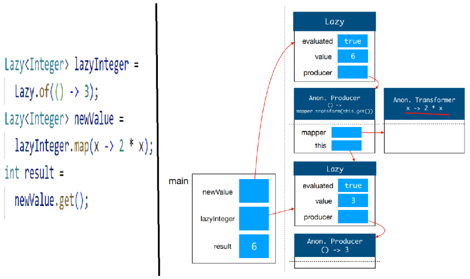
Functional Programming

1. **Pure Functions**
   - No **side effects**: **No** 1) print to the screen, 2) write to files, 3) throw exceptions, 4) change other variables, 5) modify the values of the arguments
   - **Deterministic**: Given the **same input** (can have **no input**), the function must produce the **same output**.
2. **Functional Interface**
   - (BooleanCondition<T>):
     - **Lambda example**: BooleanCondition<Integer> isPositive = x -> x > 0;
     - **Java equivalent**: Predicate<T>.
   - (Producer<T>):
     - **Lambda example**: Producer<Double> randomValue = () -> Math.random();
     - **Java equivalent**: Supplier<T>.
   - (Consumer<T>):
     - **Lambda example**: Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());
     - **Java equivalent**: Consumer<T>
   - (Transformer<U, T>): Tranform a value of type U into a value of type T.
     - **Lambda example**: Transfomer<String, Integer> stringLength = s -> s.length();
     - **Java equivalent**: Function<U,T>
   - (Combiner<S, T, R>): Combine two values of type S, T into a value of type S.
     - **Lambda example**: Combiner<Integer, Integer, Integer> multiply = (a, b) -> a * b;
     - **Java equivalent**: BiFunction<S, T, R>.
   - **Tips**
     - A **functional interface** must have **exactly one abstract method**, it **can have** any number of helpers (constants, static/default methods). It **can extend from another class**, but if the parent class has **multiple abstract method**, then the interface is **no longer a functional interface**.
     - **All** the **fields** in an **interface** are public static final (constant) by default.
3. **Method Referencing**

- **Steps to solve compile or not question**
  - Determine **the number of inputs** according to the functional interface's **abstract method** (match with the number of parameters in this method), then your lambda will be like `(x, y, ..) -> ...`, the L.H.S is the number of inputs
  - Use the rule to rewrite the normal lambda to see if 1) number of parameters matches 2) type matches (can find methods)
- **Tips**: In `A::foo`, if `foo` is an **instance method**, it will use the first input as the instance, and pass the remaining inputs as arguments. Otherwise, it will call the **class method** and pass **all inputs as arguments**.

```
Box::of              // x -> Box.of(x)
Box::new             // x -> new Box(x)
x::compareTo         // y -> x.compareTo(y)
A::foo               // (x, y) -> x.foo(y) or (x, y) -> A.foo(x,y)
```

4. **Curried Functions**: 1) Treat it as passing several parameters 2) treat it as a function that **returns another function**! e.g. `f -> x -> (f.apply(x + 0.01) - f.apply(x)) / 0.01;`.
5. **Lambda Stack and Heap**: Notice that the local variables in **main** method will be captured!



Stream

1. **reduce()**: Workflow is **result = identity → for each element in the stream; result = accumulator.apply(result, element); return result**
   - `reduce(identity, accumulator)`: the `identity` and `element in the stream` must be of the **same** type!
   - `reduce(identity, accumulator, combiner)`: the **identity may not be the same type** as the **element in the stream**. To use this safely and compatibly, must follow the **three rules**
     - **Identity Rule**: `combiner.apply(identity, i) == i`
     - **Associativity Rule**: `(x * y) * z == x * (y * z)` (**Both accumulator and combiner** should adhere to this rule)
     - **Compatibility Rule**: `combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)`
2. **map()**: produce a new stream of the transformed elements with a **one-to-one relationship**.
3. **flatMap()**: transforms **each element** into a **stream** and then **flattens** all resulting streams into a single stream, useful for working **with nested collections** or **when one element should produce multiple output elements**.
4. **filter()**: Creates a new stream that keeps **only the elements that passed the test** in `Predicate`.
5. **none/any/allMatch()**: a **boolean** method. It returns `true` only if **no/any/all** element in the stream **passes the predicate test**.
6. **sorted()**: elements according to **natural order** (small to big or ascending order) or a **provided comparator**
7. **Tips**
   - **Times evaluated by calling `get(n)`**

generate	iterate	map	flatMap
1	n	1	1

   - `sorted()` and `distinct()` are **stateful**, so should only be called on **finite streams**. Otherwise, enter infinite loop.
   - **InfiniteList**: always use `head()` and `tail()`
     - `head()`: return the **first non-filtered** element of the List.
     - `tail()`: return the remaining list, excluding the **filtered-out** elements.

Monad and Functors

1. **Monad**: A class that can be created using `of` and chained using `flatMap`. And follows the three **Monad Laws**
2. **Monad Laws**: These laws are used on **part of your lambda expression**
   - **The Left Identity Law**: `Monad.of(x).flatMap(x -> f(x))` ≡ `f(x)`. (The output of `f(x)` is by default a monad).
   - **The Right Identity Law**: `monad.flatMap(x -> Monad.of(x))` ≡ `monad`.
   - **The Associative Law**: `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` ≡ `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`
3. **Functor**: A class that has a `map` method and follows the two **Functor Laws**
4. **Functor Laws**
   - **Identity Law**: `functor.map(x -> x)` ≡ `functor`.
   - **Composition Law**: `functor.map(x -> f(x)).map(x -> g(x))` ≡ `functor.map(x -> g(f(x))`.
5. **Tips**
   - A class/type can be **both** monad and functor!
   - Every **Monad is a Functor**, but **not every Functor is a Monad**.
   - **Trace through the program step by step** to see if the certain class/type logic follows the three Monad Laws anot.
   - When using whatever monad/functor law, find the **part of the expression** that you are using that law, and replace it with the output of that law.
   - To find the ultimate proof, try comparing **what you want** with **what you have** now. And then slowly slowly prove it.

Parallel Stream

1. **Creation**: 1) call `.parallel()` on a **stream** or 2) call `.parallelStream()` on a **collection**.
2. **After Creation**: The original stream is divided into several **substreams** or **threads** to run concurrently.
3. **When can stream be parallelized**: The stream **should not have**
   - an operation with a **side effect**
   - an operation that **interferes with the stream data source.**
   - an operation that is **stateful**, meaning depending on elements processed before, e.g. `sorted()`, `distinct()`, `limit()`
4. **Tips**
   - **All parallel** programs are **concurrent**, but **not all concurrent** programs are **parallel**.
   - Having **multiple cores/processors** is a **prerequisite** to running a program in **parallel**.

Threads and CompletableFuture

1. **Threads**
   - **Decide which Thread you are in**
     - Find the position of the method call: `Thread.(whatever)`
     - If it is **inside a lambda or Runnable** which is passed to `new Thread(...)`, you're in that **new thread**.
     - If you're outside the `new Thread(...)`, e.g. in the `main` method, you're usually in the `main` thread.
   - **Pause a thread**: Done by calling `Thread.sleep(ms)`, it will pause the **thread you're in** (Use the method above to find out) for *ms* seconds.
2. **CompletableFuture**
   - **Rule of thumb**
     - `CF(f).then(g)` means: **start** g **only after** `f` **has been completed**.
       - Examples: `thenRun`, `thenCombine` – combine, `thenApply` – map, `thenCompose` – flatMap, etc.
     - `static CF.async(g)` means: **start** g **on a new thread**
       - Examples: `supplyAsync`, `runAsync`, etc.
     - `CF(f).then...async(g)` means: **start** g **only after** `f` **has been completed, but use a new thread.**
       - Examples: `thenRunAsync`, `thenApplyAsync`, etc.

     Difference between `run` and `supply`: `run` executes a **void function** while `supply` executes a **function with a return value**.
   - **Creation**: 1) `completedFuture(value)`, 2) `runAsync(Runnable)`, `supplyAsync(Supplier)` 3) Rely on other CompletableFutures, use `cf = anyOf(cfs)/allOf(cfs)`, which means the target `cf` will complete when **any/all** of the `cfs` complete.
   - **Get the result**: 1) `get()`, will throw **checked exceptions** that must be handled 2) `join()`, will not and is **usually preferred**. These two methods are **synchronous calls**, will block the **main thread** until the `cf` finishes.
   - **When is CompletableFuture complete**
     - For `CompletableFuture<T> cf`:
       - `cf = CompletableFuture.supplyAsync(s)`: **complete** after `s.get()`
       - `cf = CompletableFuture.runAsync(r)`: **complete** after `r.run()`
       - Applies regardless of nested CompletableFutures in s/r
     - For `CompletableFuture<...> cf = CompletableFuture.completedFuture(f)`:
       - **complete** immediately after creation
3. **Tips**
   - `System.out.println` is a **synchronous** method call!
   - **Thread**: There is **no sequence** of which thread will be executed first. So, be always careful, there may be lots of possibilities!
   - **CompletableFuture**: If **no** `.join()` or `.get()` is called after the CompletableFuture, the output may have **many possibilities**, a.k.a **non-determinstic output**.
   - **CompletableFuture**: Possible output with `anyOf()` is a bit **tricky**, fully utilize your **exhaustive thinking**.

Fork and Join

1. **Working Principles**
   - Each thread has a deque of tasks.
   - When a thread is idle, it checks its deque of tasks.
     - If the deque is **not empty**, it picks up a task at the head of the deque to execute (e.g., invoke its `compute()` method).
     - Otherwise, if the deque is **empty**, it picks up a task from the ***tail*** of the deque of another thread to run. This is a mechanism called *work stealing*.
   - When `fork()` is called, the caller (target) adds itself to the ***head*** of the deque of **the executing thread**. This is done so that the most recently forked task gets executed next, similar to how normal recursive calls.
   - When `join()` is called, several cases might happen.
     - If the subtask (target, same for the follow) to be joined **hasn't been executed**, this subtask will be **popped out first**, and then its `compute()` method is called and the subtask is executed.
     - If the subtask to be joined **has been completed** (some other thread has stolen this and completed it), then the result is read, and `join()` returns.
     - If the subtask to be joined has been stolen and is being executed by another thread, then the current thread either finds some other tasks to work on from its **local deque**, or steals another task from **another deque**.
2. **Tips**
   - The `fork(), compute(), join()` order should form a **palindrome** and there should be **no crossing**.

```
left.fork();  // >-----------+
right.fork(); // >--------+  | should have
return right.join() // <--+  | no crossing
      + left.join(); // <-----+
```

   - `task.compute()` is just a **normal method call**, this target `task` won't be added to the current worker's task dequeue.
   - When dealing with **work stealing** problem, always write down the **content** of the worker to-be-stolen's **task dequeue**, and its **task at the tail** will be stolen!